# A Network-Failure-Tolerant Message-Passing System For Terascale Clusters

Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E Rasmussen, L. Dean Risinger and Mitchel W. Sukalski

Los Alamos National Laboratory[*]

Advanced Computing Laboratory

MS-B287

Los Alamos, NM 87545 USA

lampi-support@lanl.gov

March 13, 2003

[*]Los Alamos report LA-UR-02-892. Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Project support was provided through ASCI/PSE and the Los Alamos Computer Science Institute.

**Abstract**

The Los Alamos Message Passing Interface (LA-MPI) is an end-to-end network-failure-tolerant message-passing system designed for terascale clusters. LA-MPI is a standard-compliant implementation of MPI designed to tolerate network-related failures including I/O bus errors, network card errors, and wire-transmission errors. This paper details the distinguishing features of LA-MPI, including support for concurrent use of multiple types of network interface, and reliable message transmission utilizing multiple network paths and routes between a given source and destination. In addition, performance measurements on production-grade platforms are presented.

## 1 Introduction

High performance computing has traditionally been the domain of the supercomputer: expensive, special purpose, vector and/or parallel systems from specialist computer companies (*e.g.*, Cray YMP, Cray T3x, Meiko CS-2, Thinking Machines CM5). The hardware and software developed by these vendors had to be designed to meet strict performance and fault-tolerance criteria demanded by their customers (a few large corporate or governmental organizations).

Recently, supercomputer-level performance has become achievable using large clusters of commodity-based systems (*e.g.*, Beowulf clusters). While promising excellent price–performance, these systems pose a new set of challenges to the system designer, namely:

- Obtaining the required performance by integrating disparate hardware and software;
- Achieving acceptable levels of fault tolerance from commodity hardware; and
- Cost- and time-effective management of very large systems.

With regard to fault tolerance, Saltzer et. al.[1] and Stone and Partridge[2] have pointed out the need for addressing the end-to-end aspect of network data transfer. Saltzer et. al.[1] cites as an example corrupt file transfers that were the result of not having end-to-end data protection. This is often the case for data transfers over network devices that are not integrated with the cluster's memory subsystems, such as networks that plug into the PCI bus. In such systems, some components, such as the data transfer between network interface cards (NIC), may have sufficient data protection. The transfers between system memory and NICs, however, are not protected, and so require a high level protocol to provide end-to-end reliability. Stone and Partridge[2] studied real TCP/IP internet data traces and showed a fairly high rate of corrupt end-to-end data tranfers, errors that went undetected by the TCP or UDP checksum. They argue that applications should provide end-to-end data checks.

The situation is aggravated for large-scale applications in which very large amounts of data are transferred in the course of a single application run. In these scenarios TCP/IP performance is often inadequate, with O/S bypass protocols being used to achieve acceptable levels of application performance.

We have observed application performance degradation as large as a factor of two when comparing the performance using TCP/IP versus O/S bypass over HIPPI-800, on an Origin2000 cluster. While using TCP/IP as a low level data transfer protocol provides some level of end-to-end data protection, O/S bypass protocols often leave this up to the application to provide this type of data protection.

The Message Passing Interface (MPI) standard [3], the *de facto* standard inter-process communication API for scientific applications, does not address the issue of corrupt data delivery, dropped data, or mis-delivered data. It assumes that data always arrives intact at the correct destination. Most current implementations either ignore this problem altogether, or use TCP/IP, at a significant performance cost, as the means to protect data transfers. Others, however, have recognized the need to provide some sort of higher level data integrity checks when using using O/S bypass protocols to implement MPI. These include protocols such as GM from Myricom [4], and Portals from Sandia National Laboratory [5].

In addition to the issue of corruption during data communication, other sources of failure appear in terascale clusters. With the sheer number of components that make up a terascale cluster, system component failure becomes not only possible, but probable for an application which runs for a sufficient amount of time. These include things such as processors, fans, interconnects, switches, network interface cards, and more. In discussing such failures, we use a three level taxonomy.

1. The lowest level deals with link level failures. These include data corruption and data loss, and in LA-MPI are addressed at a network device specific level.

2. The middle level deals with transient and permanent network failures, and is implemented in LA-MPI above the network device layer.

3. The upper level deals with more catastrophic failures, such as process loss.

This classification is a little different than that used by Bosilca [6] to analyze ways of dealing with MPI fault tolerance. Bosilica's taxonomy focuses on which bit of software deals with failure, the low level messaging library, cooperation of the library and the application, or the application.

LA-MPI is an implementation of MPI in which we address fault tolerance at all of these levels. LA-MPI implements version 1.2 of the standard, and is integrated with ROMIO [7] for MPI-IO version 2 support. LA-MPI (a) reliably delivers messages in the presence of I/O bus, network card and wire-transmission errors; (b) survives network card and path failures (when the operating system survives) and guarantees delivery of in-flight messages after such a failure; (c) supports the concurrent use of multiple types of network interface; and (d) implements message striping across multiple heterogeneous network interfaces, and striping of message fragments across multiple homogeneous network interfaces.

There have been a number of research efforts attempting to incorporate network and process fault tolerance into message passing systems. To date most of this research has been done in the context of checkpoint/rollback recovery systems. One of the first efforts to incorporate fault tolerance into MPI was CoCheck tuMPI [8] from Technischen University Munich. CoCheck used the Condor [9] library to checkpoint and then if necessary restart and rollback the MPI job. This system's main drawback was the need to checkpoint the entire application, which could be prohibitively expensive in terms of time and scalability for large applications (like those that would run on a terascale cluster). Another effort, Starfish MPI [10], is similar in operation to CoCheck, and also operates at an upper level. However, Starfish uses its own systems to checkpoint jobs, and does not rely on a flush message protocol to handle communications. Starfish uses "atomic" group communications protocols based on the Ensemble system [11]. A third upper level approach is the FT-MPI [12] effort from the University of Tennessee-Knoxville. FT-MPI handles fault tolerance at the MPI communicator level, and lets the application developer decide what course of action they wish to take. The application may decide to shrink, rebuild or abort the communicator depending on the type of fault.

An assumption implicit in many of these systems is that the underlying communication layer (this includes the reliability protocol built on top of the media) delivers data error-free. Stone and Partridge among others have shown that this assumption is not valid. The use of a reliability protocol like TCP is no guarantee of end-to-end data integrity. LA-MPI's novelty is based on the end-to-end system design philosophy of Saltzer, Reed and Clark [1], in which the only guarantee of reliability occurs when the end-points of the communication have agreed on the validity of the data sent. At the same time LA-MPI provides excellent performance. Only when end-to-end reliability is assured can the more complex challenges of process check-pointing, rollback, and recovery be addressed.

## 2 Architecture

The LA-MPI architecture is outlined in Figure 1. To simplify the discussion we divide the design into three parts: the *MPI interface layer,* the *Memory and Message Layer* (MML) and the *Send and Receive Layer* (SRL). The MPI layer provides a thread-safe MPI 1.2 [3] compliant interface for compatibility with existing applications. The MML provides policy-driven management of physical and logical resources. The SRL performs the low-level data communication.

### 2.1 MPI Layer

As mentioned above, the MPI layer implements an MPI 1.2 compliant API, the *de facto* message-passing standard for scientific applications. Although LA-MPI is designed in a modular fashion, and the MPI layer could be replaced by alternative API wrapper layers, it should be emphasized that the need to provide a complete and efficient MPI implementation led to specific architectural design choices in the lower layers of LA-MPI. In part this is a reflection of the complexity of the MPI standard.

It is worth noting that MPI promises applications the correct delivery of data. To date, most MPI implementations assume that a lower-level protocol or transport provides this guarantee. This is valid if the transport is, for example, a TCP/IP driver, or shared memory. There are, however, many examples of high-performance interconnects with OS bypass software support where hardware-level reliability is not adequately handled. LA-MPI provides an implementation of MPI which guarantees correct data delivery to the application in such circumstances.

## 2.2 Memory and Message Layer

The Memory and Message Layer (MML) is composed of a *memory manager*, a set of *network paths*, and a *path scheduler*.

The memory manager controls all memory (physical and virtual), including the process private memory, shared memory, as well as "network memory," such as memory on the NIC. Memory is managed in several pools, both process private and process shared, which are used for the allocation of buffers of various types using a free-list allocation strategy to optimize buffer reuse. Special attention is paid to memory locality issues on NUMA (non-uniform memory architecture) multiprocessor systems: shared memory pools are set up for each process so that a request can be made for memory "close" to the process which will access it most.

A network path is a homogeneous transport abstraction used to encapsulate the properties of different network devices and protocols. A path controls access to one or more network interface cards (NICs), Within a path there may be several independent "routes" corresponding to physical NICs. Currently implemented paths include UDP/IP (over any physical transport), HIPPI-800 and Quadrics Elan3, with on-going development for Myrinet 2000. Messaging between processes on the same host is handled by a special shared memory "network" path which uses additional optimizations.

An example may clarify this concept. The Nirvana cluster at LANL is a cluster of 16 SGI Origin 2000 128 processor systems, linked together with 4 independent HIPPI-800 switches, and gigabit ethernet. LA-MPI provides three paths: shared memory, HIPPI-800, and UDP/IP over gigabit ethernet. The HIPPI-800 path comprises the entire HIPPI-800 interconnect, using multiple (4) independent sub-paths or routes between a given pair of end-points.

The path scheduler "binds" a specific message between given source and destination processes to a particular path, so that different messages between the same end-points may use different paths. Though still under development, the intention here is that the routing and scheduling algorithms can be selected at compile time or run time, and may be a default or user-written module. An algorithm might schedule messages across paths according to message properties (*e.g.* size, destination, *etc.*), and/or use statistics-based heuristics. Depending on message size and available routes, a single message may be striped across several routes.

In the Nirvana cluster described above, whole messages may be scheduled across different paths (HIPPI-800 and UDP/IP), while fragments of a single message may be striped across up to four different routes.

The MML architecture implements several desirable features:

- *Message striping* across several network paths, thereby increasing network utilization and performance, is straightforward. By message striping we mean, sending different messages between the same end points via different network paths, such as an O/S bypass over HIPPI-800 path and a UDP/IP path.

- *Message-fragment striping* across several routes within a single network path is possible when a path comprises more than one network interface. E.g., a single message is fragmented and sent along multiple network paths, such as over multiple HIPPI-800 nics and switches. At the destination, this message is reassembled.

- *Reliability* is implemented within the path abstraction. The path is responsible for breaking an outgoing message into one or more fragments, and reassembling incoming fragments into complete messages. During fragmentation and reassembly, a path specific "checksum" function is used to verify correct transmission. For HIPPI-800 we use a a 32 bit data checksum, and in the UDP/IP implementation we rely on the network checksum provided by this protocol and assume that any data we receive is correct. The Send and Receive layer (SRL - see below) uses the "checksum" when deciding how to proceed. If data corruption is detected the entire fragment is retransmitted. The detailed protocol for this process is discussed in section 3.

- *Resilience* to network device failure is a function of the path scheduler. In case of a network route failure, evidenced by many failed message transmissions, the path scheduler will attempt to "rebind" outstanding messages to another valid (and functional) route between the source and destination processes. This route may use the same path or it may be assigned to a different path. Future messages will not be bound to the failed route. The ability to "fail-back" to the first route (corresponding to the case of a temporarily unavailable network device) is also planned. This work in progress.

## 2.3 Send and Receive Layer

The Send and Receive Layer (SRL) is responsible for sending and receiving message fragments, and is highly network dependent.

The physical path message fragments fall into two categories: those that require the network (off-host) and those that do not (on-host). On-host messages are simple copies through shared memory. Off-host messages are handled by the Network Communication module, where the message fragments are sent via physical resources associated with the path to which the message is bound.

As mentioned above, the SRL layer also handles message fragmentation and reassembly. Message reassembly occurs in the order in which the fragments are received, and in systems with multiple routes between a pair of end-points out-of-order fragment arrival is a common event, and is handled correctly by this layer. This layer also handles the arrival of duplicate message fragments which can occur with timer based data retransmission.

Finally, the SRL layer also handles the in-order delivery required by the MPI standard. Data that arrives out-or-order is queued for later processing, as is unexpected data.

Figure 2 illustrates the steps in a typical off-host point-to-point communication. For simplicity, assume the message consists of a single fragment. When the user specifies a send, the MML determines the appropriate path and fragments the message. The fragment is sent to the destination while the source waits for an acknowledgment. For a multiple fragment message, the fragments are sent in parallel as long as resources are available, and the acknowledgments can be received in any order. If the fragment was not received properly (determined either by a negative acknowledgment or time out), the fragment is retransmitted. If the fragment was received properly, the old fragment is freed. A more detailed description of the architecture is available [13].

## 3 Reliability

### 3.1 Why include a reliability layer?

"Reliable" network protocols and devices are often designed to one set of criteria, and deployed in environments that fail to respect these design assumptions. For example, high-performance NICs are sometimes capable of assuring reliable data transfer between NICs by doing reliable transport protocol (*e.g.* TCP) processing on the NIC itself. Unfortunately, this reliability guarantee is negated if the NIC itself is plugged into an unreliable I/O bus. Given the complexity of modern computers, the net result is that in large cluster environments application to application reliability is often quite difficult to achieve, and its lack almost impossible to diagnose.

Why not use the nearly ubiquitous TCP transport protocol (executed on the main CPUs)? The answer is, in a word, performance. TCP/IP-based messaging has relatively high latency due to the maintenance of connection state that allows its heuristics to operate in environments from noisy 56Kbps dialup modems to gigabit ethernet LANs. High performance cluster environments, however, are usually implemented with modern local and system area networks that are capable of supporting very low latency in the range of 3-30 microseconds from NIC to NIC. A well-performing message library must provide reliability over a range of network devices (*e.g.* Myrinet, Quadrics, gigabit ethernet, *etc.*) using protocols with minimal impact on latency (*e.g.* UDP/IP, VIA, *etc.*). In order to minimize latency in these environments, LA-MPI uses its own lightweight protocol to provide reliability over a diverse set of network technologies.

The reliability layer in LA-MPI shares a number of attributes with other reliability layers (most notably TCP) including the use of watchdog timers, checksums, and sequence numbers to check for duplicate, lost, or corrupt data [14, 15]. Unlike most TCP implementations, the LA-MPI reliability layer is implemented in user space much like the reliability protocol in Globus-Nexus [16]. Figure 2 shows the basics of the LA-MPI reliability protocol.

The protocol uses sender side retransmission to achieve the desired level of reliability. Messages are fragmented into fixed-sized chunks, or “fragments”. Each fragment is assigned a sequence number (out of a monotonically increasing sequence of 64-bit values) and a timestamp, and the number of times a given fragment has been sent is updated each time the fragment is sent. Retransmission is scheduled on a per fragment basis using a truncated exponential backoff scheme for every retransmission attempt; the backoff scheme helps protect receiver resources in the event the receiver is busy executing non-communication code.

Upon receiving the fragment, the receiving process sends either a positive acknowledgment, ACK, or a negative acknowledgment, NACK, to the sending process. ACKs can be of two types: fragment specific, and non-specific. Fragment specific ACKs are generated when the receiving process has successfully received a fragment, verified its ”checksum”, and copied its data into application-specified memory. Non-specific ACKs are generated when a duplicate fragment is received. They contain information about the largest in-order sequence numbers seen from a sending peer for data that has been received, and for data that has been successfully received and delivered to the receiving application (*i.e.* copied out of the LA-MPI library). Non-specific ACK information is piggybacked in each fragment specific ACK, and is used by the sender to delay retransmission of fragments that have been received but have not yet been acknowledged by a fragment-specific ACK or NACK.

NACKs are generated when the data received is corrupt (*i.e.* fails ”checksum” verification upon being copied to application memory). Upon receiving a NACK, the sender will arrange to retransmit the data.

Figures 3 and 4 illustrate the overhead of providing end-to-end reliability in LA-MPI. The overhead is small, and ranges between 2 to 15% for x86 Linux and 2 to 3% for SGI Origin2000. In LA-MPI's reliable UDP/IP implementation, this low overhead is due to our reliance on UDP/IP's checksum, and the relatively infrequent rate (every several seconds) at which we check to see if a fragment needs to be retransmitted. On HIPPI-800 on the Origin2000, the overhead of reliability is minimized by calculating checksums concurrently with data copying.

## 4 Performance

In this section we present LA-MPI performance data on a simple ping-pong benchmark and a representative scientific application, CICE (the Los Alamos Sea Ice Model). For comparison, we also present data from other available MPI implementations where this is available, notably Argonne National Laboratory's reference MPI implementation, MPICH version 1.2.3, and SGI's Message Passing Toolkit version 1.5.2 (MPT). We choose to compare our results with MPICH because it is the most widely known implementation, and many vendor implementations orginated from it. SGI's implmentation is used on the Origin because it is the vendor's implementation, and is tuned for it's systems.

Performance was measured on Nirvana, LANL's 16-machine cluster of SGI Origin 2000 (O2K) machines (128 250 MHz R10K processors per machine) running IRIX 6.5, and representative commodity machines, namely

Dell Precision 610 PCs running RedHat Linux 7.1 and 7.2 (dual 550MHz Pentium III processors for on-host testing and a single processor for off-host testing). Off-host communication on Nirvana was accomplished over the HIPPI-800 (100 MB/s peak performance) interconnect using user-level operating system bypass support, and using UDP/IP over gigabit ethernet (1 Gbps peak performance). Off-host communication on the Dells was accomplished using UDP/IP over a 100 Mbps switched ethernet. On-host communication in both environments uses anonymous shared memory.

While current performance can be characterized as good to excellent (while providing services other MPI libraries do not), we expect these numbers and other performance metrics to improve as we continue optimizing the library. A more comprehensive performance evaluation is under way.

## 4.1 Ping-Pong Latency and Bandwidth

Table I shows the zero-byte half-round-trip message latency for LA-MPI, SGI MPT and MPICH from (Argonne National Lab) in micro-seconds.

As this table indicates LA-MPI has very good zero-byte latency. On the Origin2000, the on-host latency of LA-MPI is 7.0 micro-seconds, 8% worse than SGI's MPT, but nearly three times better than MPICH. Over HIPPI-800 LA-MPI's latency is about 8% higher than SGI's MPT. MPICH has no implementation for this device. LA-MPI's (UDP/IP) and MPT's (TCP/IP) latency over gigabit ethernet are virtually identical, while MPICH (TCP/IP) has a latency that is about 11% higher.

On the Dell's LA-MPI shared memory latency is 2.3 micro-seconds, an order of magnitude better than MPICH, but over switched ethernet MPICH is currently about 8% better than LA-MPI.

Figures 5 and 6 compare the bandwidth achieved by LA-MPI, MPT and MPICH at various message sizes.

Comparing on-host (shared memory) bandwidths on the Origin 2000, at small message sizes MPT performs better than LA-MPI, but by the time one reaches larger messages size LA-MPI outperforms MPT. For example at 256 byte messages, MPT runs at 19 MB/s, and LA-MPI at 17 MB/s, at 8 KB MPT gets about 101 MB/s and LA-MPI gets about 87 MB/s, but at 1 MB MPT gets 135 MB/s, but LA-MPI runs at 145 MB/s. MPICH does not perform as well as LA-MPI and SGI's MPT, with bandwidth of 9.9 MB/s, 75 MB/s, and 80 MB/s, respectively.

Similar results are obtained for shared memory communications on the Dell when we compare LA-MPI and MPICH, with LA-MPI showing significantly better bandwidths than MPICH. At 256 bytes LA-MPI is running at 59 MB/s, at 8 KB bytes at 75 MB/s, peaking out at 64 KB message sizes at a bandwidth of 169 MB/s, and at 512 KB the rate is 93 MB/s. MPICH's performance is 9.6 MB/s, 75 MB/s, 131 MB/s, and 86 MB/s, respectively.

For off-host communication, comparing LA-MPI and SGI's MPT bandwidths, we see that for small messages the two are comparable, but for larger messages LA-MPI has significantly better performance, since LA-MPI stripes fragments of a single message across several HIPPI-800 routes (4 in this case). At 256 bytes LA-MPI's bandwidth is 1.5 MB/s, at 8 KB 18.8 MB/s, at 128 KB 86 MB/s, and at 1 MB 135 MB/s. For SGI's MPT the bandwidths are 1.4 MB/s, 20 MB/s, 57 MB/s, and 73 MB/s, respectively. The relatively large differences (about 60%) already evident at 128 KBytes can be understood, since the striping is done in chunks of 16 KB, and at the size of 128 KB each HIPPI-800 route is already handling 2 message fragments.

Figure 7 illustrates the benefits of message-fragment striping, that is, sending message fragments in parallel over different NICs. LA-MPI chops HIPPI-800 messages into 16 KB fragments, and the benefit of striping is obviously not evident until the message length is more than one fragment.

Low latency is achieved in a variety of ways. Shared memory latency is low, because we use lockless queues in non-threaded operation, and memory management costs are minimized through the use of a simple free-list strategy. For cross-host communication, the checksum cost is proportional to the length of data transmitted, and therefore does not have a significant impact on latency as seen in table I. Simultaneously, LA-MPI achieves high bandwidth by fragmenting messages which allows the overlap of sender and receiver memory copies. For shared memory, copying is also overlapped with tag matching.

## 4.2 CICE: the Los Alamos Sea Ice Model

CICE [17] is a widely used production code for efficiently modeling sea ice in a fully coupled atmosphere-ice-ocean-land global climate model. CICE is a community model developed by scientists at LANL, the National Center for

Atmospheric Research (NCAR), and other universities.

CICE is also a good benchmark program for evaluating MPI implementations. It is written in Fortran 90 using a wide assortment of MPI features including point-to-point communication (MPI_ISEND/MPI_IRECV), broadcasts, reductions, MPI groups, and MPI datatype operations. In addition, CICE uses a fairly even distribution of message sizes with slightly more very small and very large messages (see Table II). This removes strong biases toward particular message sizes when evaluating performance.

CICE is typically run on eight processors on the SGI Origin 2000 at LANL. Figure 8 shows the performance of CICE for 64 and 128 time steps for MPICH, SGI MPT, and LA-MPI. Performance with LA-MPI (99.18 seconds) is within 3% of SGI MPT (96.47) for the 64 time step case, and within less than 2% of SGI MPT for the 128 time step case (191.85 seconds and 188.78 respectively). MPICH ran 18% and 20% slower than LA-MPI, respectively.

The point-to-point communications times are also interesting to compare with LA-MPI (8.02 seconds) taking 5% less time than SGI's MPT (8.43 seconds) and 69% less time than MPICH (13.54 seconds) at the 64 time step case. For the 128 time step case these differences are 27%, and 200%, respectively, with LA-MPI taking 14.05 seconds, SGI's MPT 17.82 seconds, and MPICH 28.18 seconds.

## 5 Conclusions

In this paper, we have given a brief overview of LA-MPI, the Los Alamos Message Passing Interface, a message-passing system for terascale clusters. Such clusters will be composed of hundreds or thousands of individual commodity-based machines connected by hundreds or thousands of network interfaces over hundreds or thousands of cables. Each individual component of the system not only adds capability but also points of failure.

LA-MPI was designed with the assumption that terascale clusters are unreliable. In particular, the increasing functionality of hardware, especially with respect to data integrity, does not eliminate the need for additional software to ensure end-to-end reliability. LA-MPI's novelty is that it provides end-to-end reliability in a high performance message-passing system without significant overhead on a wide variety of network transports and devices.

With the number of system failures expected to increase with cluster size, applications must be prepared to deal with these failures. We have taken the first steps in providing *resilience* for applications running on such clusters; applications can continue through network failures as long as there is at least one physical path between source and destination processors.

LA-MPI also offers the possibility to enhance performance relative to existing message-passing systems by implementing message striping across multiple heterogeneous network interfaces, and message-fragment striping across multiple homogeneous network interfaces.

## 6 Future Work

LA-MPI is still in active development. We have recently made a port to Compaq's Tru64 UNIX to add to our existing Linux and IRIX support. Quadrics' Elan3 network interface is now supported under Tru64, and Myrinet 2000 network support under Linux is being implemented. Full automatic network path failover is in development, and future performance optimization work will address scalability issues.

In addition to these efforts, LA-MPI is part of a larger project aimed at providing complete application resilience, or *run-through*. The Cluster Research Lab in the Advanced Computing Laboratory at LANL has a number of projects that will together enable an application to run-through to completion despite hardware failures. This is particularly important for applications at LANL and other DOE laboratories that run for weeks to months before getting an "answer." LA-MPI will ultimately be integrated with Supermon [18], a cluster monitoring system that will predict failures based on vital hardware statistics such as CPU temperature and fan speeds. Applications running on processors or nodes that are predicted to fail will be migrated off to a healthy node via the BProc migration facility [19]. LA-MPI will be enhanced to support process migration.

## Acknowledgments

The authors would like to thank former members of the LA-MPI team (then called *User-Level Messaging*) including Pete Beckman, Steve Karmesin, Ling-Ling Chen, MaryDell Nochumson (for her efforts in regression testing), and members of the LANL BlueMountain support team. The authors would like to thank Rob Aulwes for his cornucopia of useful suggestions.

## References

[1] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[2] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *SIGCOMM*, pages 309–319, 2000.

[3] Jack J. Dongarra and David Walker. MPI: a standard message passing interface. *Supercomputer*, 12(1):56–68, January 1996.

[4] http://www.myri.com/.

[5] http://www.cs.sandia.gov/cplant/.

[6] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes.

[7] Rajeev Thakur, William Gropp, and Ewing Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Mathematics and Computer Science Division, Argonne National Laboratory, October 1997. ANL/MCS-TM-234.

[8] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[9] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing System*, pages 108–111. IEEE Computer Society Press, 1988.

[10] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[11] Graham E. Fagg, Keith Moore, and Jack J. Dongarra. Scalable Networked Information Processing Environment (SNIPE). *Future Generation Computer Systems*, 15(5–6):595–605, 1999.

[12] G. Fagg and a Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *EuroPVM/ MPI User's Group Meeting 2000 ,Springer-Verilag, Berlin, Germany, 2000*, 2000.

[13] Robbie T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, and Mitchel W. Sukalski. LA-MPI: The Design and

Implementation of a Network-Fault-Tolerant MPI for Terascale Clusters. In *Submitted to 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[14] J. Postel. Transmission Control Protocol. Internet Engineering Task Force, RFC 793, 1981.

[15] W. R. Stevens. *TCP/IP Illustrated, Volume 2; The Implementation.* Addison Wesley, Reading, 1995.

[16] A. Denis. Variable reliability protocol in Globus-Nexus. Technical report, Information Science Institute (ISI), University of Southern California, 1999.

[17] E. C. Hunke and W. H. Lipscomb. CICE: the Los Alamos sea ice model. Technical Report LA-CC-98-16, Los Alamos National Laboratory, 1999.

[18] Ron Minnich and Karen Reid. Supermon: High performance monitoring for linux clusters. In *The Fifth Annual Linux Showcase and Conference*, November 2001.

[19] Erik A. Hendriks. BProc: The Beowulf distributed process space. In *16th Annual ACM International Conference on Supercomputing*, 2002.

| Platform (network) | LA-MPI | SGI MPT | MPICH |
|---|---|---|---|
| O2K (shared mem) | 7.0 | 6.5 | 19.9 |
| O2K (Hippi800) | 155.3 | 143.5 | N/A |
| O2K (IP) | 526.7 | 525.6 | 586.0 |
| i686 (shared mem) | 2.3 | N/A | 23.5 |
| i686 (UDP/IP) | 132.8 | N/A | 123.5 |

Table I: Zero-byte latency in micro-seconds for various message-passing libraries

| *message size in bytes* | *number of messages* |
|---|---|
| $< 100$ | 21786 |
| 101-1000 | 18410 |
| 1001-10001 | 18410 |
| 10001-100000 | 18410 |
| $> 100000$ | 26090 |

Table II: Number of messages by size range in CICE for point-to-point communication. The distribution of message sizes is fairly even with slightly more very small and very large messages.

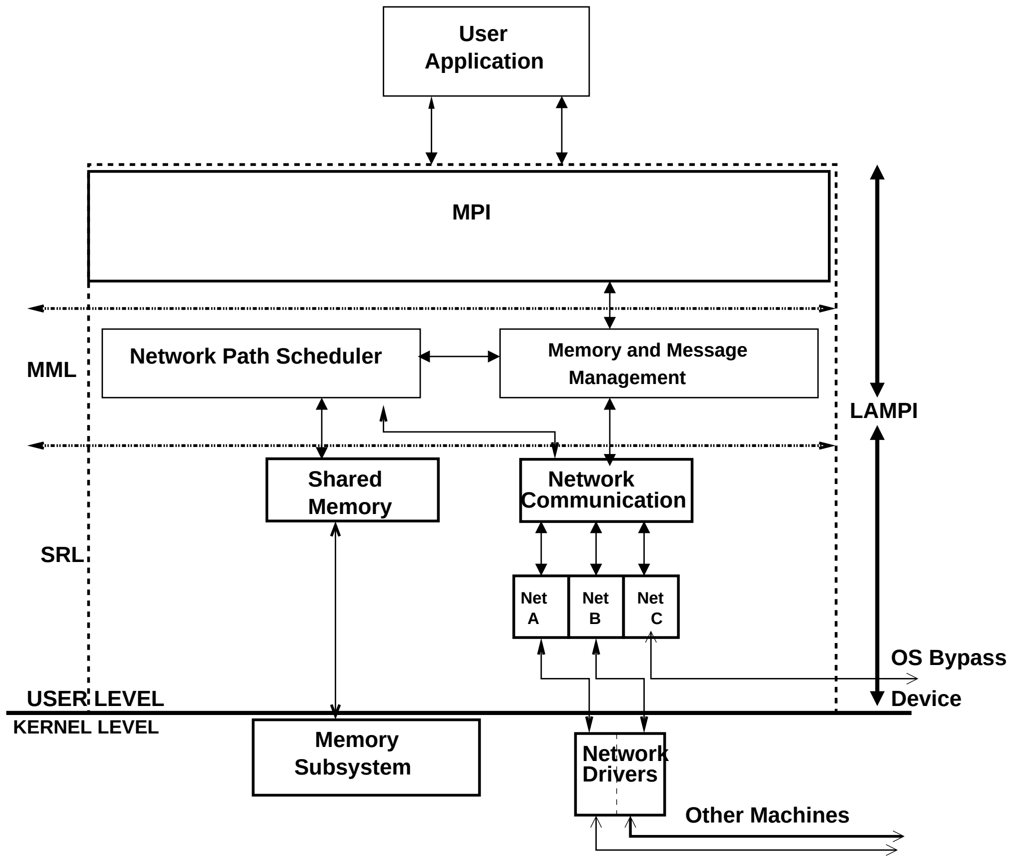

Figure 1: LA-MPI architecture overview

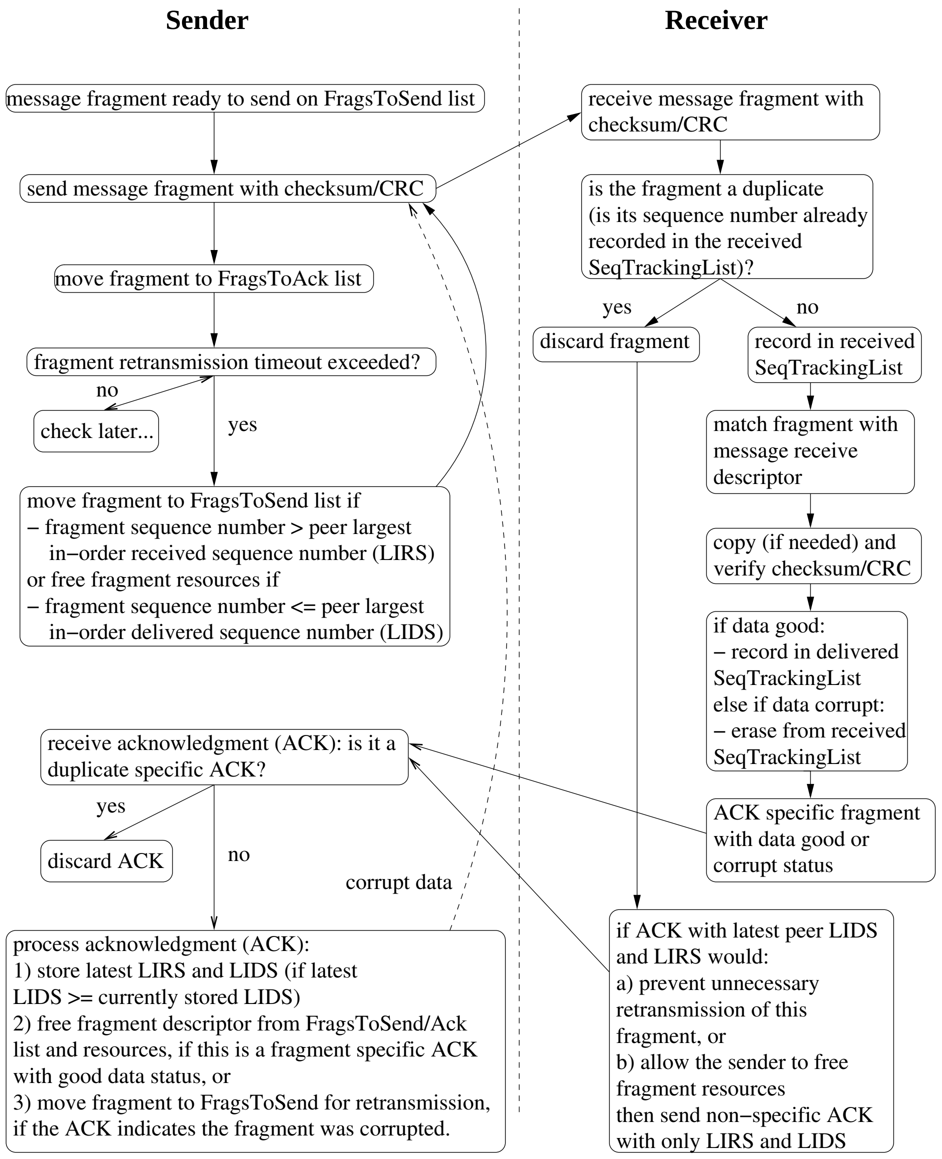
Sender
Receiver
message fragment ready to send on FragsToSend list
send message fragment with checksum/CRC
move fragment to FragsToAck list
fragment retransmission timeout exceeded?
no
check later...
yes
move fragment to FragsToSend list if
– fragment sequence number > peer largest in–order received sequence number (LIRS)
or free fragment resources if
– fragment sequence number <= peer largest in–order delivered sequence number (LIDS)
receive acknowledgment (ACK): is it a duplicate specific ACK?
yes
discard ACK
no
corrupt data
process acknowledgment (ACK):
1) store latest LIRS and LIDS (if latest LIDS >= currently stored LIDS)
2) free fragment descriptor from FragsToSend/Ack list and resources, if this is a fragment specific ACK with good data status, or
3) move fragment to FragsToSend for retransmission, if the ACK indicates the fragment was corrupted.
receive message fragment with checksum/CRC
is the fragment a duplicate (is its sequence number already recorded in the received SeqTrackingList)?
yes
no
discard fragment
record in received SeqTrackingList
match fragment with message receive descriptor
copy (if needed) and verify checksum/CRC
if data good:
– record in delivered SeqTrackingList
else if data corrupt:
– erase from received SeqTrackingList
ACK specific fragment with data good or corrupt status
if ACK with latest peer LIDS and LIRS would:
a) prevent unnecessary retransmission of this fragment, or
b) allow the sender to free fragment resources
then send non–specific ACK with only LIRS and LIDS

Figure 2: Point-to-Point off-host communication for a single fragment message. For a multiple fragment message, the fragments are sent in parallel as long as resources are available, and the acknowledgments can be received in any order.

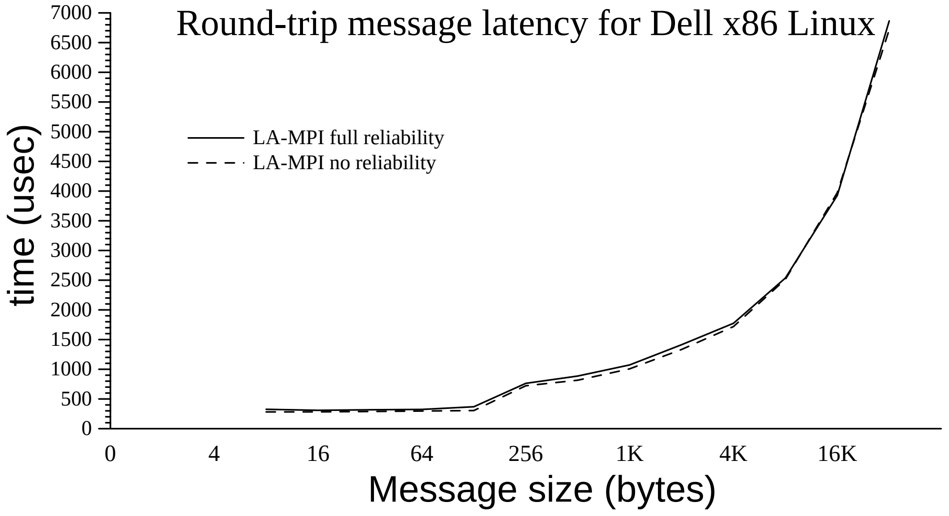

Figure 3: Comparison of full reliability LA-MPI to LA-MPI with reliability turned off for the i686 Linux.

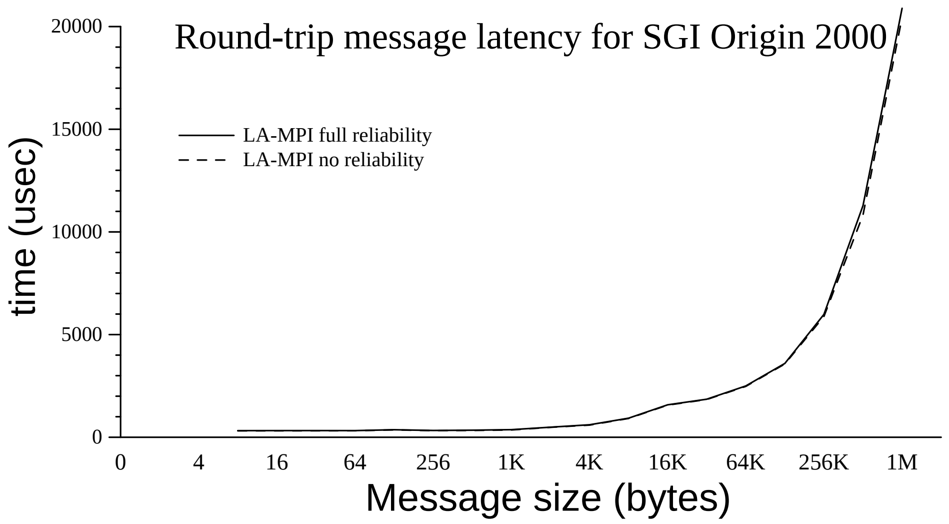

Figure 4: Comparison of full reliability LA-MPI to LA-MPI with reliability turned off for the SGI Origin 2000.

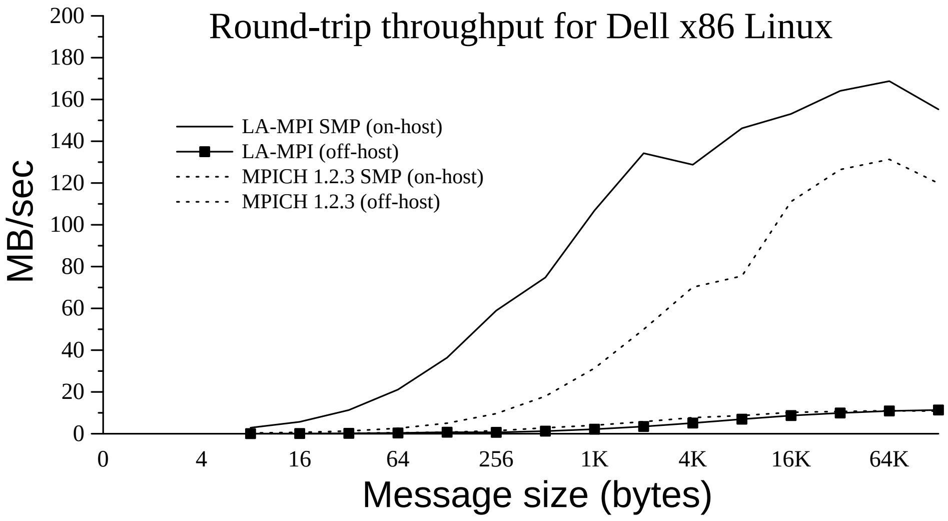

Figure 5: Comparison of round-trip message throughput for Dell PC x86 Linux.

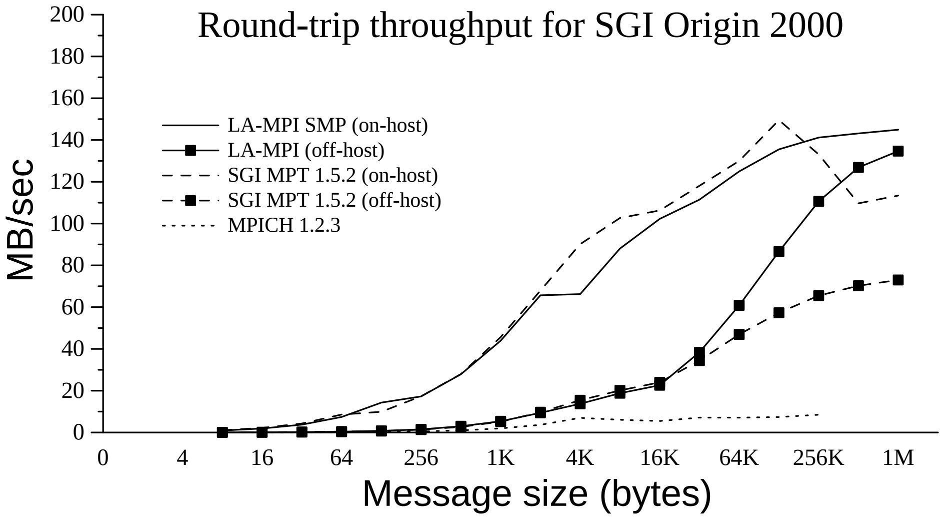

Figure 6: Comparison of round-trip message throughput for the SGI Origin 2000.

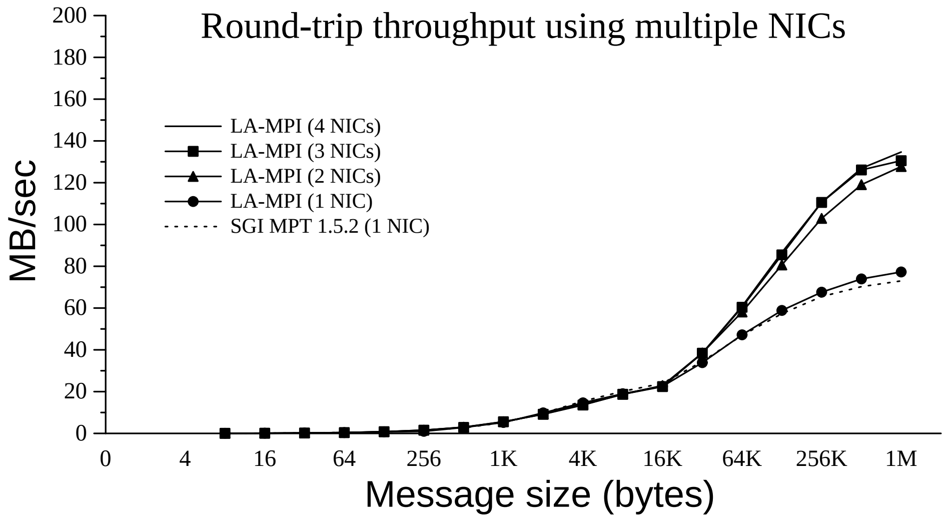

Figure 7: Round-trip throughput with message striping.

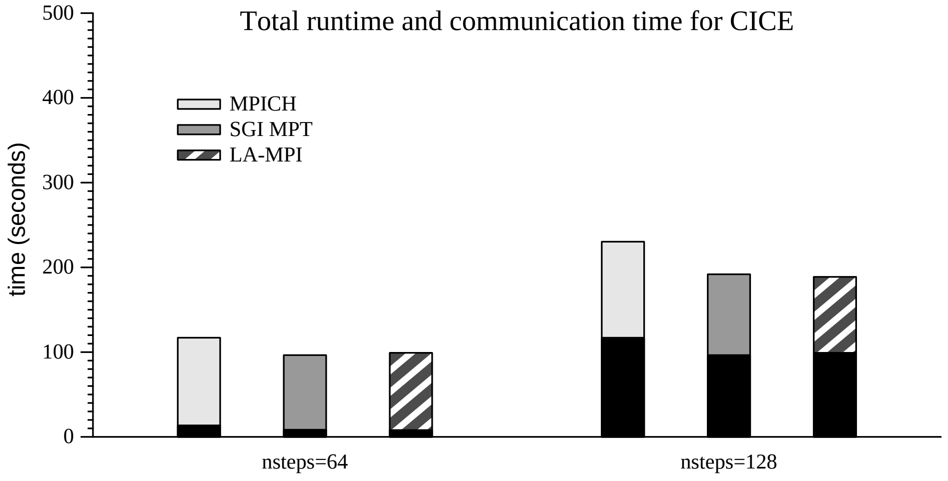

Figure 8: CICE run time using MPICH, SGI MPT, and LA-MPI for 64 and 128 time steps. The dark portion of the bars indicate time spent in communication.